

Please type a plus (+) inside this box ☐

PTO/SB/29 (12/97)

Approved for use through 09/30/00, OMB 0651-0032
Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

**UTILITY
PATENT APPLICATION
TRANSMITTAL**

(Only for new nonprovisional applications under 37 CFR 1.53(b))

Attorney Docket No.	9344-0004-999	Total Pages	36
First Named Inventor or Application Identifier			
Zhen Zia			
Express Mail Label No.	EL 451 593 961 US		

APPLICATION ELEMENTS
See MPEP chapter 600 concerning utility patent application contents.**ADDRESS TO:** Assistant Commissioner for Patents
Box Patent Application
Washington, DC 20231

1. ☒ Fee Transmittal Form
Submit an original, and a duplicate for fee processing
2. ☒ Specification [23 Pages]
(preferred arrangement set forth below)
- Descriptive title of the Invention
 - Cross Reference to Related Applications
 - Statement Regarding Fed sponsored R&D
 - Reference to Microfiche Appendix
 - Background of the Invention
 - Brief Summary of the Invention
 - Brief Description of the Drawings *(if filed)*
 - Detailed Description of the Invention (including drawings, *if filed*)
 - Claim(s)
 - Abstract of the Disclosure

- 6.
- ☐
- Microfiche Computer Program
- (Appendix)*

- 7.
- ☐
- Nucleotide and/or Amino Acid Sequence Submission
-
- (if applicable, all necessary)*

- a. ☐ Computer Readable Copy
- b. ☐ Paper Copy (identical to computer copy)
- c. ☐ Statement verifying identity of above copies

3. ☒ Drawing(s) *(35 USC 113)* [8 Sheets]
4. ☒ Oath or Declaration [2 Sheets]

- a. ☐ Newly executed (original or copy)
- b. ☐ Copy from a prior application (37 CFR 1.63(d))
(for continuation/divisional with Box 17 completed)
[Note Box 5 below]
- i. ☐ **DELETION OF INVENTOR(S)**
Signed statement attached deleting inventor(s) named in the prior application, see 37 CFR 1.63(d)(2) and 1.33 (b).
5. ☐ Incorporation By Reference *(useable if Box 4b is checked)*
The entire disclosure of the prior application, from which a copy of the oath or declaration is supplied under Box 4b, is considered as being part of the disclosure of the accompanying application and is hereby incorporated by reference therein.

ACCOMPANYING APPLICATION PARTS

8. ☐ Assignment Papers (cover sheet & document(s))
9. ☐ 37 CFR 3.73(b) Statement ☐ Power of Attorney
(when there is an assignee)
10. ☐ English Translation Document *(if applicable)*
11. ☐ Information Disclosure ☐ Copies of IDS
Statement (IDS)/PTO-1449 Citations
12. ☐ Preliminary Amendment
13. ☐ Return Receipt Postcard (MPEP 503)
(Should be specifically itemized)
14. ☐ Small Entity Statement filed in prior application,
Statement(s) Status still proper and desired
15. ☐ Certified Copy of Priority Document(s)
(if foreign priority is claimed)
16. ☐ Other:

17. If a **CONTINUING APPLICATION**, check appropriate box and supply the requisite information:
- ☐ Continuation ☐ Divisional ☐ Continuation-in-part (CIP) of prior application No: filed.

18. CORRESPONDENCE ADDRESS

- ☒ Customer Number or Bar Code Label 20583
(Insert Customer No. or Attach bar code label here) or ☐ Correspondence address below

NAME			
ADDRESS			
CITY	STATE	ZIP CODE	
COUNTRY	TELEPHONE	FAX	

Burden Hour Statement: This form is estimated to take 0.2 hours to complete. Time will vary depending upon the needs of the individual case. Any comments on the amount of time you are required to complete this form should be sent to the Chief Information Officer, Patent and Trademark Office, Washington, DC 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Assistant Commissioner for Patents, Box Patent Application, Washington, DC 20231.

METHOD AND APPARATUS TO PREVENT REVERSE ENGINEERING AND TAMPERING

BACKGROUND OF THE INVENTION

Field of the Invention

5 The present invention relates to methods and apparatus that can prevent, resist, or deter reverse engineering and tampering with information such as computer software or data files during both the static and dynamic states of its presence on a system.

Background of the Invention

10 Most computer software and data found on commercially available general purpose operating systems are exposed to a threat of being reverse engineered or tampered with using widely available disassembling, de-compiling, debugging, and in-circuit emulating utilities. Despite the employment of cryptographic algorithms, hardware dangles, and software encryption, software and data remain vulnerable to security attacks. Such vulnerability exists regardless whether the software and data are present on a computer
15 system in a static state (such as on a hard drive or other non-volatile storage media) or a dynamic state (such as in residence in a cache memory or main memory).

 For example, by taking advantage of the appropriate utilities, an attacker skilled in the art of computer security or security cracking can observe and re-assemble the instructions of a software program by tracing their execution image in memory. The
20 attacker can further monitor and/or alter a software program's secret operations such as its interactions with physical components of a computer. The attacker can also de-compile and analyze compiled code in the static state and then alter critical sections of the compiled code to compromise security.

25 To increase the difficulty for an attacker to observe, understand, or modify source code, companies such as Intel and Intertrust have introduced elaborate schemes of transformation or slicing of source code. The potential pitfall of these protection schemes is that they rely on the ingenuity of their designers. Attackers on the other hand similarly rely on their ingenuity to reverse engineer a protector's design. Thus, the effectiveness of the

protection system becomes an ingenuity contest between the protector and the attacker. Unfortunately, this fails to provide a scientific measure of how easy or how difficult it is for the ingenious protection mechanisms to be broken.

For these and related reasons, we assume that (1) all compiled code and data files are observable given the availability of commercial hardware and software utilities; (2) all elaborate schemes can be reverse engineered by ingenuity; and (3) attackers know the design of security schemes every bit as well as the designers. We believe the security of a protection system should be predictable and measurable. The most appropriate and reliable measure is probably the computation time and cost required to crack the protection system. In particular, true security lies in a predictable, large work factor for attackers. Such work factor should be large enough to make it humanly impossible to comprehend the protected source code and data files, and exponentially time consuming and expensive for computers to do so.

SUMMARY OF THE INVENTION

The present invention protects computer software by adding to the software large numbers of obscuring instructions selected from an obscuring code bank. Preferably, the obscuring instructions selected from the obscuring code bank are made to resemble the computer code that is obscured to achieve uniqueness of obscuration at each installation. Such obscuration can be achieved through embodiments both at the source code level and the object code level although different apparatus may need to be employed.

Preferably, the obscuring instructions are generated in functional groupings called "blocks". Advantageously, at least some of the blocks of obscuring instructions are formed from other blocks by a transformational relationship. Specifically, for any two successive blocks, C_1 and C_2 , that have a transformational relationship, the instructions in C_2 can be determined and generated by performing a mathematical transformation T on number codes associated with the instructions in C_1 . And, in general, any block C_N is generated by a series of mathematical transformations T such that $C_N = T_N(T_{N-1}(...(T_3(T_2(C_1))))...$ where C_1 is an initial block of obscuring instructions selected from the obscuring code bank. Conversely, instructions in C_1 can be determined and generated by performing the inverse mathematical transformation on number codes associated with the instructions in C_2 . Advantageously, different mathematical transformations are used between different pairs of successive blocks

and the transformations are randomly selected so as to achieve uniqueness for each set of obscuring instructions that is generated.

To enhance security, the obscured object code is encrypted and stored in the form of superblocks of concatenated blocks of code. Consequently, each block of code in a superblock can be decrypted only if the blocks of code that precede it in the superblock have previously been decrypted. This results in an obscured package of code that is resistant to analysis of the statically stored code or to any tampering while in the static state.

Advantageously, the obscured object code may also be compressed to remove some of the redundancy arising from the use of mathematical transformations to generate some of the obscuring code blocks.

In the present invention, the obscured code package is loaded into a computer's real memory block by block. A run time apparatus is employed to decrypt/decompress and generate the obscured instructions of each block starting with C_1 . The generated instructions of each block are loaded into memory by this apparatus at a dynamically determined address that is unique for each block. After loading the block, the run time apparatus switches control to the instruction block for execution. When execution of each block is completed, control is switched back to the run time apparatus to load the next block. The process continues until instructions in all the blocks are executed. The dynamic loading and execution of each block makes it virtually impossible to trace instructions that are only generated and executed in real time.

A preferred method of operating the invention to protect a sequence of computer code comprises the steps of: preparing simple obscuring instructions that are comprehensible yet require considerably more time to read and understand; injecting a large number of obscuring instructions into the sequence of computer code in an automated process to produce an obscured sequence of computer instructions that in total is humanly impossible to read and understand; compressing and/or encrypting a static image of the obscured sequence to protect against direct decompilation; and executing the obscured instructions one instruction at a time, thereby making run time trace and observation a labor intensive manual process. Preferably, the method provides a computational work factor that is exponential at least on the scale of N^3 where N is the number of obscured instructions and potentially may be as much as $e^{N \log N}$. For example, for 10,000,000 obscured instructions, it can be expected that it would take over 250,000 years on a modern PC (e.g., 500 MHZ clock rate) to locate and reverse engineer the protected sequence of computer code.

In similar fashion, the present invention also protects data files by adding to the data large numbers of obscuring data selected from an obscuring data bank. Blocks of obscuring data can also be generated that are related to each other by a mathematical transformation. The transformation can be performed either on the data itself or on number codes associated with each item of obscuring data.

To enhance security, the obscured data may likewise be encrypted and/or compressed. This may be done as part of the same process that encrypts and/or compresses the obscured object code or it may be done separately.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other objects, features and advantages of the invention will be more readily apparent from the following detailed description in which:

Fig. 1 describes a first part of the apparatus of the present invention.

Fig. 2 describes a second part of the apparatus of the present invention.

Fig. 3 describes a third part of the apparatus of the present invention.

Fig. 4 describes a fourth part of the apparatus of the present invention.

Fig. 5 illustrates the content of a data file used for run time decryption.

Fig. 6 illustrates the structure of run time components of the system in reference to a computer's execution environment.

Fig. 7 describes the process of loading and executing the first block of obscured code segments and the loading of data for the first N blocks.

Fig. 8 describes the process of loading and executing the second obscured code block and transfer of control from second block to the third block.

DETAILED DESCRIPTION OF THE DRAWINGS

In Fig. 1, a pre-processor 102 parses source code 101 to generate serialized code blocks 104 and a critical function profile 106. This process is completed on the source code level at pre-compile time. Source code 101 is typically the source code used in critical functions, such as the most crucial part to the overall security of a computer program, or source code that contains the most essential implementation details in realizing certain valuable design and other trade secrets. The present invention is primarily to protect the

critical function source code from being identified, observed, traced for execution or modified (often referred to as "patched"). Source code 101 is considered serialized in the present invention when all subroutine calls in source code 101 have been fully expanded into sequentially listed instructions in one self-contained body function/subroutine. By transforming source code 101 into the serialized code blocks 104, source code 101 is prepared for injecting obscuring code in later stage processes of the present invention.

Pre-processor 102 uses a user defined security strength 103 as an input parameter in determining the number of lines of code to be generated in each code block 104 and the total number of code blocks 104. For the highest level of security strength 103, each code block 104 will contain at most only one line of source code 101 or one instruction, and the total number of code blocks 104 is equal to or greater than the number of lines of source code in the critical function source code 101. For the lowest level of security strength 103, there could be only one code block 104 that may contain all the original source code 101 within the single block. For other levels of security strength 103, the pre-processor 102 may randomly set a number of code blocks, *NC*, and randomly determine the number of instructions, *NI*, in each code block 104.

Pre-processor 102 also generates a data set called critical function profile 106 that describes the nature of the source code contained in the critical function source code. Profile 106 is understandable to other components of the present invention and is utilized for selecting obscuring code that "looks" similar to make it difficult for an attacker to distinguish the original critical function source code 101 and the obscuring code that is injected later.

As an example of obscuring a simple instruction in a function, consider a simple C program segment where the critical operation instruction is "V1 = 1024":

```
Function1 (....) {  
...  
Int V1; //declare the variable as integer type  
V1 = 1024; //Assign value to V1  
...  
Return V1; // return the value of V1  
}
```

The objective here is to obscure the assignment operation $V1=1024$, which is simple and straight forward and will not take more than a few seconds for a person skilled at the art to understand. However, by simply injecting some obscuring code, the obscured code will be significantly more time consuming to read and comprehend. The obscuring code selected may include a number of assignment instructions and some simple calculation instructions.

```
ObsFunction1 ( ... ){
  Int V1, V2, V3, V4;

  ...
  V1 = 1024;
  V1 = V1 + 1024;
  V2 = 1024;
  V2 = V2 + 1024;
  V3 = V2;
  V3 = V3 - V2;
  V4 = 1024;
  V4 = V4 + 1024;
  V4 = V4 - V4
  V2 = V2 - 1024;
  V1 = V1 - 1024;
  V2 = V2 - 1024;
  V1 = V1 + V2 + V3 + V4;
  ...
  Return V1;
}
```

With the added obscuring instructions, it is still possible to isolate $V1$ out of $V2$, $V3$, and $V4$. However, now it will take a few minutes before a skilled person can read and identify the original critical instruction " $V1=1024$ ". The complexity of this example increases enormously when the original critical instructions are composed of 10 to 20 lines of instructions, these instructions are mixed with obscuring instructions on the order of

millions including both instructions that are similar to the instructions to be protected and instructions that are dissimilar, and the critical instructions are randomly spread across many blocks. In fact, it becomes so time consuming to read and comprehend the obscured output that it will become humanly impossible. For example, consider the difficulties involved in interpreting a sequence of code if, instead of four values V1, V2, V3 and V4, the code included 100,000 values, returned all 100,000 values and there was no indication of which value or values had any significance.

Fig. 2 depicts an obscuring code generator 203 and two predefined code banks: an obscuring code bank 204 and a transformation function bank 205. In Fig. 2, obscuring code generator 203 generates obscuring code blocks 206 that are used to protect the critical function source code 101. Illustratively, the available storage size is one million lines of code, the size of each block 206 is 125 lines of code, and there are 8000 blocks of code. Obscuring code generator 203 uses the critical function profile 106, security strength 103, storage size 201, and execution time 202 as input parameters for generating obscuring code from the two predefined code banks.

Obscuring code bank 204 is a database that contains program instructions previously created through a manual, automated, or a combination of manual and automated process. Associated with each instruction is a unique numeric code. Thus, each numeric code identifies an obscuring instruction and the entire set of numeric codes identifies the entire set of obscuring instructions. Advantageously, each numeric code may simply be the memory address at which the instruction is stored in the obscuring code bank. Each code block 206 is a subset of the obscuring instructions available in obscuring code bank 204 and the instructions in block 206 can be identified by the numeric codes associated with those instructions.

The program instructions in bank 204 comprise a large pool of instructions that are often built upon expertise and experience of the database designers. A large number of them resemble the most frequently used instructions in commonly used programming languages although a significant portion of them are purely random code without predefined profile. As indicated in the example given above, obscuring code may perform a function (e.g., returning the value $V1 = 1024$) that is useful to the critical function code to be protected but do so in a way that is very inefficient. Indeed, it may be spectacularly inefficient. Alternatively, as suggested by operations such $V4 = V4 - V4$, the code may be functional but may do nothing more than perform an operation and later perform the inverse

of the operation so as to produce no effect other than obscuring the code to be protected. Other examples of obscuring code may have nothing do with the operation of the code to be protected but will still have to be deciphered because an attacker will not know which code is relevant and which is not.

5 The presence of the obscuring instructions injected into the critical function source code has a direct impact on any attacker's ability to understand or modify the correct instructions in order to compromise the system's integrity. As an example, a segment of 10 lines of instructions that performs certain essential functions of a software application is serialized, distributed into multiple code blocks and mixed with 1,000,000 lines of
10 obscuring code instructions. Using modern day microcomputers, one can assume that the typical CPU clock speed is beyond 500 MHZ. At such speeds, it takes a microcomputer no more than 8 milliseconds to execute all 1,000,000+ instruction provided the instructions perform relatively simply calculations, assignments and minimal I/Os. While the computational overhead is relatively low, the job for an attacker to understand and modify the key parts of these instructions is an insurmountable task. Even though the task of
15 obtaining and observing the instructions is already difficult, let's assume an attacker can capture all 1,000,000 plus instructions and can observe and analyze them. This number of instructions amounts to 16,667 pages of printout on regular letter sized paper. Assuming the attacker can read at the speed of 3 minutes per page and work for an average of 8 hours a
20 day, it will take him over 100 days just to finish reading the content in order to reach a shallow understanding of the instructions. As a practical matter, careful examination and much more time will ordinarily be required to identify the original 10 lines of critical instruction out of the 1,000,000 obscuring instructions.

25 Furthermore, if a certain set of instructions is of even higher importance in a software application, more obscuring instructions can be injected. Consider the example of injecting 10,000,000 obscuring instructions. A modern computer can process these instructions within 80 milliseconds. However, the attacker will be challenged with a total of 166,667 pages of printout, and over three years just to finish reading them casually. The challenge is practically equivalent to finding a small needle in the Atlantic Ocean, which is
30 humanly impossible to do.

 Current and future advances in computer microprocessor are rapidly accelerating. Today, CPUs that work at one GHz have been announced and CPUs that work at over 500 MHZ are commonplace. The faster a CPU can process instructions, the more obscuring

instructions can be injected to protect critical functions of applications, and consequently, the harder it becomes for attackers to understand, identify or modify the protected instructions.

Additionally, there are no currently known pattern recognition algorithms that can automatically parse, understand and locate critical instructions found in a large number of obscured instructions. Due to the largely random nature of the obscuring instructions, the pattern recognition task can be highly difficult. By adding more elaborate transformations and slicing of original code in combination with the large number of injected obscuring code, development of a pattern recognition algorithm can be made even more difficult. It is reasonable to assume the computational complexity of such an algorithm is at least as high as $O(N^3)$, where N is the number of instructions, with the possibility of being even as high as $O(e^{N \log N})$. In the case of 10,000,000 or more instructions, using the $O(N^3)$ estimate, one can expect the computation can take as long as 250,000 years on a 500MHz CPU modern day personal computer.

On the other end of the scale, it is evident that substantial protection can be achieved using far fewer obscuring instructions than 1,000,000. Even 10,000 lines of obscuring instructions represent a day's effort to read and typically much more time to understand. How much more time is a function of the intricacy of the code. As a practical matter with appropriate obscuring instructions, we believe it is reasonable to assume that it would take several months' effort to reach sufficient understanding of 10,000 such instructions to be able to identify and understand the operation of critical instructions embedded in such obscuring instructions. In some applications, several months' time is enough protection. As will be apparent, greater amounts of protection can be achieved with increasing numbers of lines of obscuring instructions. With 100,000 lines of obscuring instructions, we estimate the amount of time for one individual to reach an understanding of the operation of the critical instructions to be several years, which is often the length of time that a software product enjoys commercial success. In such circumstances, 100,000 lines of obscuring instructions may be enough protection.

The transformation function bank 205 is a database previously created to contain mathematical functions that are one-to-one mappings from Set A to Set B and their inverse functions that are one-to-one mappings from Set B to Set A. Associated with each transformation is a unique numeric code. Thus, each numeric code identifies a transformation and the entire set of numeric codes identifies the entire set of

transformations. Advantageously, each numeric code may simply be the memory address at which the instruction is stored in the transformation code bank. Preferably, Sets A and B are sets of numeric codes and the transformation T satisfies the following relationships:

$$B = T(A)$$

and

$$A = T^{-1}(B)$$

where T^{-1} is the mathematical inverse function of T. Examples of T might be increment the value of A by 10 or multiply the value of A by 3; and the corresponding inverse functions would be decrement the value of B by 10 and divide the value of B by 3.

Obscuring code generator 203 applies the transformations obtained from the transformation function bank 205 to the numeric codes associated with the obscuring instructions obtained from the obscuring code bank 204 to produce more obscuring instructions. In particular, generator 203 produces blocks of obscuring code 206. The first of these blocks is generated by generator 203 by selecting obscuring instructions from code bank 204. Additional blocks are generated by selecting transformations 210 from transformation function bank 205, applying these transformations to the numeric codes associated with the obscuring instructions found in a previously generated block of obscuring instructions so as to generate a set of transformed numeric codes and forming new blocks 206 of obscuring instructions using the instructions identified by the transformed numeric codes. Preferably, the transformations are selected randomly. The selected transformation functions are represented by elements 208 in Fig. 2 and their inverses by elements 211.

In the event a transformation generates a numeric code that is outside the range of numeric codes, the generated numeric code “wraps around” as in modulus arithmetic so as to generate a numeric code that is within range.

In the embodiment of the invention shown in Fig. 2, the transformations are concatenated so that any block C_N is generated by a series of mathematical transformations T such that $C_N = T_N(T_{N-1}(...(T_3(T_2(C_1)))...))$, where C_1 is an initial block of obscuring instructions selected from the obscuring code bank. By concatenating the transformations, it is possible to generate an enormous number of different transformations while storing only relatively few transformations in the transformation function bank 205. Alternatively,

each block can be generated from the first block of obscuring instructions using a single transformation function instead of the concatenated set of functions.

The use of transformations to generate additional blocks of obscuring instructions makes it possible to generate enormous numbers of additional obscuring instructions while allowing the system to compress and encrypt these instructions. To someone trying to understand the instructions, a block of instructions generated by a transformation of associated numeric codes can be every bit as difficult to understand as the original block of instructions. However, the transformed block can be represented simply by the transformation which can be represented by its numeric address in the transformation code bank. Thus, while it would require 125 numeric codes associated with instructions in code bank 204 to represent a first block of 125 obscuring instructions, a single numeric code associated with a transformation in function bank 205 can be used to generate from the first code block another 125 numeric codes associated with instructions in code bank 204 to represent a second block of 125 obscuring instructions and so on for additional blocks of obscuring instructions. Moreover, if the correspondences between the numeric codes and the obscuring instructions and the numeric codes and the transformation functions can be kept secret, the instructions may also be encrypted.

It should be noted, however, that there are also computational costs involved in generating the additional blocks of code using the transformations. As a result, a typical practice is for generator 203 to produce several different blocks of obscuring code by selecting instructions from code bank 204 and then generate from each of these blocks of obscuring code several additional blocks of obscuring instructions by selecting transformations 210 from the transformation function bank 205.

The composition of the first code block, the number of code blocks, the size of each code block, the number of obscuring instructions per line of code to be obscured, and the compression ratio to be maintained are determined by generator 203 from the critical function profile 106, security strength 103, storage size 201 and execution time 202.

Fig. 3 depicts an obscuring code injector 301, run time apparatus 302 and an obscuration compiler 308. In Fig. 3, obscuring code injector 301 combines the serialized code blocks 104 and obscuring code blocks 206 with run time apparatus 302 to create a pre-compilation obscured image 307. Obscuration compiler 308 uses the pre-compilation program image 307 as input to create an obscured object level image 312.

Run time apparatus 302 comprises the necessary programming instructions to load blocks of machine level code into a computer's memory for execution and to transfer execution control from one code block to another. A code locator 303 locates one or more blocks of programming instruction from a data file that will be described later in Fig. 6. A decryptor 304 decrypts the code block located by the code locator 303 into plain text machine level code in preparation for execution. A code loader 305 loads the decrypted code block into memory and starts the actual execution of the instructions. A control handler 306 hands over control of execution from the current code block to the next one in queue as soon as the current code block's execution is finished.

Obscuring code injector 301 injects the run time apparatus 302 comprising elements 303, 304, 305, and 306 into the serialized critical function source code blocks 104 and the obscuring code blocks 206 to minimize the possibility for the serialized critical function source code to be observed. As a result, image 307 comprises multiple collections of blocks 302, 104, and 207. At this stage, the pre-compilation obscured image 307 is ready to be compiled into object code. The obscuration compiler 308 is applied to pre-compilation obscured image 307 to create object level code blocks 309, 310, and 311 in correspondence to blocks 302, 104, and 206. Each collections of a block 309, block 310, and block 311 is referred to as an object level block O_i 312.

Obscuration compiler 308 is a special purpose apparatus that augments a regular compiler by preserving the transformation constraints. With a regular compiler, the transformation function T for adjacent obscuring code blocks 206 would be lost once the source code is compiled into object level code using a regular compiler. However, obscuration compiler 308 implements the processing logic to preserve such transformation function even after object level code is created for the source code. Specifically, if the functional constraint exists between block C_1 206 and block C_2 206 and can be defined as follows (same as in Fig. 2):

$$C_2 = T_2(C_1)$$

and

$$C_1 = T_2'(C_2)$$

where T_2 and T_2' are transformation functions and inverse transformation functions for C_1 and C_2 , then, the obscuration compiler ensures that the corresponding object level code blocks $O_{C1} 311$ and $O_{C2} 311$ satisfy the following constraints:

$$O_{C2} = T_2(O_{C1})$$

and

$$O_{C1} = T_2'(O_{C2}).$$

The implementation of a compiler that preserves the transportation information in this way will be known to those skilled in the art. By so preserving the transformation information, the transformation functions can be applied to the object level code to achieve compression, if desired.

In Fig. 4, an encryption processor 401 takes object level blocks 312 as input, and encrypts them in a recursive chain fashion. The encryption process is applied to all object level code blocks 312 starting with O_N which includes blocks $O_{LN} 309$, $O_{EN} 310$, and $O_{CN} 311$ and ending with O_2 which includes blocks $O_{L2} 309$, $O_{E2} 310$, and $O_{C2} 311$. The process is not applied to object code block O_1 which includes blocks O_{L1} , O_{E1} , and O_{C1} . The output of each stage i of encryption processor 401 is $D_i 402$. The output of each stage except stage 2 is applied as an input to the encryption processor of the next stage.

In general, each encryption processor P_i scrambles and thereby encrypts object level code block $O_i 312$ and the output of D_{i+1} of the previous processor in accordance with an algorithm specified by a key. Advantageously, a different scrambling algorithm is used for each encryption processor P_i and the key that specifies the algorithm is inserted in clear text in the output D_i . This encryption process ensures that the output data file is encrypted and can not be directly de-compiled statically. Because the blocks are encrypted as they are compressed, attackers cannot directly decompile the data files to obtain the entire obscuring and critical function instructions. These characteristics force attackers to trace the execution of the system in this invention during run time as the only feasible means to observe the obscuring instructions.

Fig. 5 illustrates the data file constructed at the end of the obscuring process. The final data file contains essentially $O_{L1} 309$, $O_{E1} 310$, $O_{C1} 311$ and D_2 . D_2 in turn contains the scrambled form of D_3 , and D_3 contains D_4 , so on and so forth. Code blocks obscured in this fashion are protected against any direct de-compilation or disassembling attempts, because

the contents of the data file are no longer recognizable for utilities that do not understand the specific format and de-compression process.

Because the number of obscuring instructions can be in the millions, it is also desirable to incorporate data compression technology in the encryption processors 401. In the case where blocks of obscuring code are generated by mathematical transformations from a first block of obscuring code generated from obscuring code bank 204, substantial compression can be achieved simply by representing each block in terms of the first block of obscuring code and the numerical codes representing the transformations used to generate the block. Since obstruction compiler 308 ensures that the object level code that represents the obscuring instructions is related by just such a series of transformations, such compression is achievable by replacing the object level code blocks O_{Ci} 311 with the transformations that are used to generate these blocks. In addition, the numeric codes that identify these transformations can readily be scrambled and thereby encrypted at the same time as the encryption processor scrambles and encrypts object level code blocks O_{Li} 309 and O_{Ei} 310.

In particular, in a preferred embodiment of the invention that both encrypts and compresses the object level code, encryption processor P_N 401 scrambles O_{LN} 309 and O_{EN} 310 and the numeric code representing transformation T_N in accordance with an algorithm specified by a key N . These scrambled values and the clear text value of the key N constitute output D_N 402. Compression is achieved by representing O_{CN} 311 in terms of the scrambled numeric code representing transformation T_N .

Subsequently, D_N 402, a numeric code representing transformation T_{N-1} , a key $N-1$ and the next set of object level code blocks O_{N-1} are used as inputs for encryption processor P_{N-1} 401. At this step, encryption processor P_{N-1} 401 scrambles O_{LN-1} 309, O_{EN-1} 310 and D_N 402, and a numeric code representing transformation T_{N-1} . These scrambled values and the clear text value of key $N-1$ constitute output D_{N-1} 402. Again, compression is achieved by representing O_{CN-1} 311 in terms of the scrambled numeric code representing transformation T_{N-1} .

The compression and scrambling process continues for all the object level code blocks in 312 in the same fashion for sequence number $N-2$, $N-3$, $N-4$... except for O_{Li} 309, O_{Ei} 310, and O_{Ci} 311.

Once the process is complete, O_{Li} 309 can retrieve key₂ so as to de-scramble D_2 402 at runtime to retrieve T_2 , O_{L2} 309, O_{E2} 311 and D_3 ; and O_{C2} 311 can be recreated using O_{Ci}

311 and the unscrambled numeric code for transformation T_2 . However, until D_2 is descrambled and O_{L2} is executed, D_3 remains undistinguishable. And similarly, until D_3 is descrambled and O_{L3} is executed, D_4 remains undistinguishable; and so on. Such constructs ensure that all the code blocks are only observable when the scrambled data blocks O_{L1} , D_2 , D_3, \dots, D_{N-1} and D_N are scrambled and executed at runtime.

Fig. 6 depicts a typical microcomputer system architecture and the execution model of the run time apparatus of the present invention. A data file 602 is stored on the computer's hard disk 601. The data file is loaded into the computer's real memory 603 at run time through the computer's main bus system 604. The actual memory space required to execute the code blocks contained in the data file is allocated separately and is illustrated at 605.

Fig. 7 illustrates how the run time execution process starts with the first set of code blocks being loaded into memory and executed.

All executions of the code blocks are conducted within the memory address space indicated as 605. As the first step, O_{L1} 701 is loaded at memory address L_1 711. This address will remain unchanged for all future $O_{L2}, \dots, O_{LN-1}, O_{LN}$ code blocks. CodeLoader 702 of O_{L1} executes within this space to allocate a dynamic memory location at address E_1 710. It is important that address E_1 710 be dynamically assigned to ensure that the execution process of the code blocks can not be automatically traced at a fixed address using conventional or commercially available tools. Because of the dynamic nature of this address allocation, address E_2 for the next set of code blocks O_{L2} can not be determined until the active instructions of O_{L1} de-scramble O_{L2} .

At address E_1 , the runtime image of a series of code blocks is loaded and executed, including O_{E1} 704, O_{C1} 705, Get(key₂) 706, Decrypt (O_{L2}) 707, Load (O_{L2}) 708, and "Jump To Address L" 709. O_{E1} 704 and O_{C1} 705 are the mixed instruction blocks that contain both the original instructions in critical function source code 101 and the obscuring instructions for the first code block. The execution of instruction blocks 704 and 705 is the most essential action at this stage.

Get(key₂) 706 is the instructions that retrieves the encryption key₂ so that O_{L2} can be decrypted and loaded into Address L. Decrypt(O_{L2}) 707 is the set of instructions that actually decrypts and creates the executable O_{L2} code blocks. "Load (O_{L2}) at Address L" 708 loads the decrypted O_{L2} instructions into the static memory address L 711 ready for the next step of processing. "Jump to L" 709 hands the control of execution to the instructions

loaded at address L 711. At this stage, the essential functionality has been completed for step one and the system is ready to load and execute the next set of code blocks O_{L2} 309, O_{E2} 310, and O_{C2} 311.

In Fig. 8, O_{L2} 801 has been decrypted and loaded into the static memory address L 711. A code locator 802 locates the compressed and scrambled code for O_{E2} 807 and O_{C2} 808 from the .DAT data file cache 606 and retrieves them for decryption and decompression. Decryptor 803 executes the actual decryption and decompression of the retrieved code blocks. A code loader 804 dynamically determines a memory address E_2 806, allocates the necessary space, and loads the decrypted code blocks into it for execution. A control handler 805 transfers control of execution to the instructions loaded at E_2 806.

O_{E2} 807 and O_{C2} 808 contain the true instructions within the original critical function source code 101 and the obscuring code blocks. They are first executed as the most essential functionality of this step. Subsequent code segments 809, 810, and 811 are similar to the apparatus described in Fig. 7, and are executed to retrieve O_{L3} , load O_{L3} into the static memory space at address L, then transfer execution control over to the instructions in that memory space.

The execution of subsequent sets of code blocks, O_{L4} 309, O_{E4} 310, O_{C4} 311, O_{L5} 309, O_{E5} 310, O_{C5} 311, ..., follow the same process as described above until all code blocks are loaded in memory and executed.

Because the run time apparatus in this invention allows dynamic loading and execution of the blocks in data file, virtually an arbitrary number of obscuring instructions can be executed as long as execution overhead limit permits. Furthermore, because every block of instruction is executed at a dynamically assigned memory address, it makes tracing execution of these blocks a challenging task. Without highly specialized hardware devices, locating the address where a block of instructions is loaded in memory is virtually impossible. These characteristics of the runtime system ensure that obtaining and observing instructions in memory using tracing techniques are laborious and time consuming to the extent of being humanly impossible without the support of highly expensive and specially designed hardware devices.

The method and system described in the present invention can be applied to any digital material that includes an executable component. Whenever a software application includes implementation of highly valuable technology or other trade secrets, respective programming instructions can take advantage of the obscuring capability of the current

invention. For computer security related products, the present invention offers these products robust anti-trace and anti-decompilation protection for programming instructions that are most vulnerable and critical in the products.

5 Additionally, an embodiment of the present invention can enable different sets of obscuring programming instructions to be injected for each different protected product, user desktop computer, or user identification. High performance back end server systems can be optimized to extract obscuring instructions from the obscuring code bank specifically and differently according to the input of a machine id, user id, or other uniquely identifying parameters. Such capability to prevent any potential compromise of security can be
10 generically applied to other products, customers, or machines. It can ensure the same amount of computational resource is required to crack each product or machine across a product line or customer line.

As indicated, in the same fashion as the invention is used to protect critical software, the invention may also be applied to the protection of critical data by hiding the critical data amid vast quantities of obscuring data generated from an obscuring data bank. Additional
15 quantities of obscuring data may also be generated by transforming the obscuring data using a transformation function bank. The obscured data may likewise be encrypted and/or compressed either as part of the same process that encrypts and/or compresses the software or independently thereof.

20 In conclusion, the present invention makes it possible to protect critical programming instructions and/or data by injecting a large amount of obscuring instructions and/or data to the extent that observing and understanding of the obscured instructions and/or data is not humanly feasible. The apparatus and system of the present invention facilitates compression of obscuring instruction and/or data and the runtime execution of the
25 obscuring instructions and/or data so that neither direct de-compilation nor real time tracing of the obscuring instructions and/or data can be achieved without the use of extensive and expensive computing resources only affordable by large organizations over an extraordinary time span.

It is claimed:

1. A method for protecting a sequence of computer code comprising the steps of:

preparing obscuring instructions;

injecting a large number of obscuring instructions into the sequence of computer code in an automated process to produce an obscured sequence of computer instructions that in total is humanly impossible to read and understand; and

encrypting a static image of the obscured sequence to protect against direct decompilation.

2. The method of claim 1 further comprising the step of executing the obscured instructions one instruction at a time, thereby making run time trace and observation a labor intensive manual process.

3. The method of claim 1 wherein the obscuring instructions are identified by codes further comprising the steps of:

generating a first set of obscuring instructions having a first set of codes associated therewith;

transforming the first set of codes associated with the first set of obscuring instructions into a second set of codes; and

generating a second set of obscuring instructions identified by the second set of codes.

4. The method of claim 3 wherein the codes are numeric codes and a mathematical transformation is performed on the numeric codes of the first set to produce the numeric codes of the second set.

5. The method of claim 1 wherein the obscuring instructions are identified by numeric codes further comprising the steps of:

generating a first set of obscuring instructions having a first set of numeric codes associated therewith;

performing a mathematical transformation on the numeric codes of the first set to produce a second set of numeric codes; and

generating a second set of obscuring instructions identified by the second set of numeric codes.

6. The method of claim 5 further comprising the step of compressing the static image by recording in the static image a record of the transformation used to generate the second set of obscuring instructions.

7. The method of claim 1 further comprising the step of compressing the static image.

8. The method of claim 1 wherein the obscured sequence of computer instructions is organized into a sequence of blocks of computer instructions and the steps of encrypting a static image comprises the steps of:

encrypting a first block to form a first encrypted output;

encrypting a second block and the first encrypted output to form a second encrypted output; and

encrypting a third block and the second encrypted output to form a third encrypted output.

9. The method of claim 8 further comprising the step of compressing each block as it is encrypted.

10. A method for protecting a data file comprising the steps of:

preparing obscuring data;

injecting a large number of obscuring data into the data file in an automated process to produce an obscured sequence of data that in total is humanly impossible to read and understand; and

encrypting a static image of the obscured sequence to protect against direct decompilation.

11. The method of claim 10 wherein the obscuring data are identified by codes further comprising the steps of:

generating a first set of obscuring data having a first set of codes associated therewith;

transforming the first set of codes associated with the first set of obscuring data into a second set of codes; and

generating a second set of obscuring data identified by the second set of codes.

12. The method of claim 11 wherein the codes are numeric codes and a mathematical transformation is performed on the numeric codes of the first set to produce the numeric codes of the second set.

13. The method of claim 10 wherein the obscuring data are identified by numeric codes further comprising the steps of:

generating a first set of obscuring data having a first set of numeric codes associated therewith;

performing a mathematical transformation on the numeric codes of the first set to produce a second set of numeric codes; and

generating a second set of obscuring data identified by the second set of numeric codes.

14. The method of claim 13 further comprising the step of comprising the static image by recording in the static image a record of the transformation used to generate the second set of obscuring data.

15. The method of claim 10 further comprising the step of compressing the static image.

16. The method of claim 10 wherein the obscured sequence of data is organized into a sequence of blocks of data and the steps of encrypting a static image comprises the steps of:

encrypting a first block to form a first encrypted output;

encrypting a second block and the first encrypted output to form a second encrypted output; and

encrypting a third block and the second encrypted output to form a third encrypted output.

17. The method of claim 16 further comprising the step of compressing each block as it is encrypted.

18. The method of claim 10 further comprising the steps of:
generating a first set of obscuring data; and
performing a mathematical transformation on the first set of obscuring data to produce a second set of obscuring data.

19. Apparatus for protecting a sequence of computer code with obscuring instructions, said apparatus comprising:
means for automatically injecting a large number of obscuring instructions into the sequence of computer code to produce an obscured sequence of computer instructions that in total is humanly impossible to read and understand; and
means for encrypting a static image of the obscured sequence to protect against direct decompilation.

20. The apparatus of claim 19 further comprising means for executing the obscured instructions one instruction at a time, thereby making run time trace and observation a labor intensive manual process.

21. Apparatus for protecting a sequence of computer code with obscuring instructions comprising:
an obscuring instruction bank, for storing obscuring instructions each of which is identified by a code;
a transformation function bank for storing transformation functions;
a generator for generating blocks of obscuring instructions by selecting obscuring instructions from the obscuring instruction bank and transformation functions from the transformation function bank, said transformation functions being used by the generator to transform the codes that identify selected obscuring instructions so as to generate other codes that identify other obscuring instructions that are then selected by the generator.

22. A method of executing a plurality of critical instructions that are obscured by obscuring instructions and stored along with encrypted loading instructions and decryption keys, said method comprising the steps of:

loading into a first memory address a first block of loading instructions,
executing the loading instructions to allocate a first dynamic memory address,
loading and executing at the first dynamic memory address at least one critical instruction that is obscured by obscuring instructions,
retrieving a first decryption key,
decrypting a second block of loading instructions using the decryption key,
loading into said first memory address the second block of loading instructions,
executing the loading instructions to allocate a second dynamic memory address, and
loading and executing at the second dynamic memory address at least one more critical instruction that is obscured by obscuring instructions.

23. The method of claim 22 comprising the steps of:

retrieving a second decryption key,
decrypting a third block of loading instructions using the second decryption key,
loading into said first memory address the third block of loading instructions,
executing the loading instructions to allocate a third dynamic memory address, and
loading and executing at the third dynamic memory address at least one more critical instruction that is obscured by obscuring instructions.

Abstract

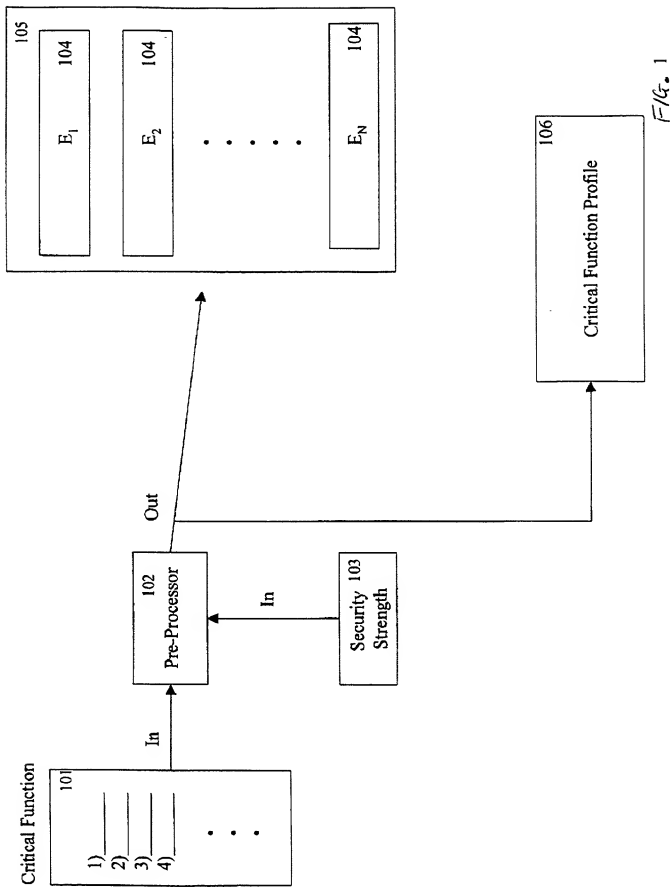
A method and apparatus are described for protecting critical computer software and/or data with a large amount of obscuring instructions and or data to the extent that observing and understanding the obscured instructions and/or data is not humanly feasible.

5 In a preferred method of obscuring software, a bank of obscuring instructions is prepared, a large number of obscuring instructions are selected from the bank and injected in the software code to be protected and a static image of the obscured sequence of code is encrypted and/or compressed. At execution, the obscured instructions are executed one at a time to make run time tracing a labor intensive process.

10

09552951.042000

Serialized Code Blocks



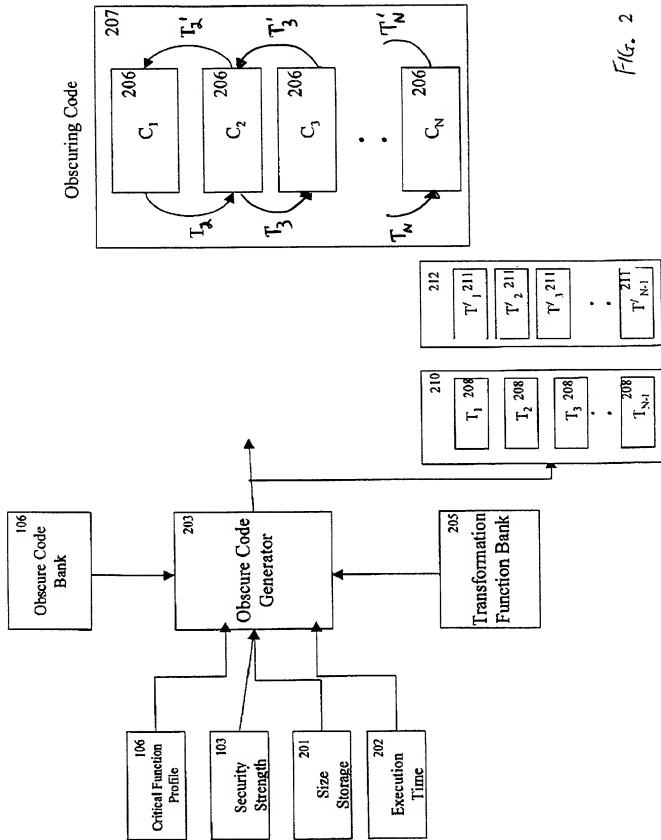
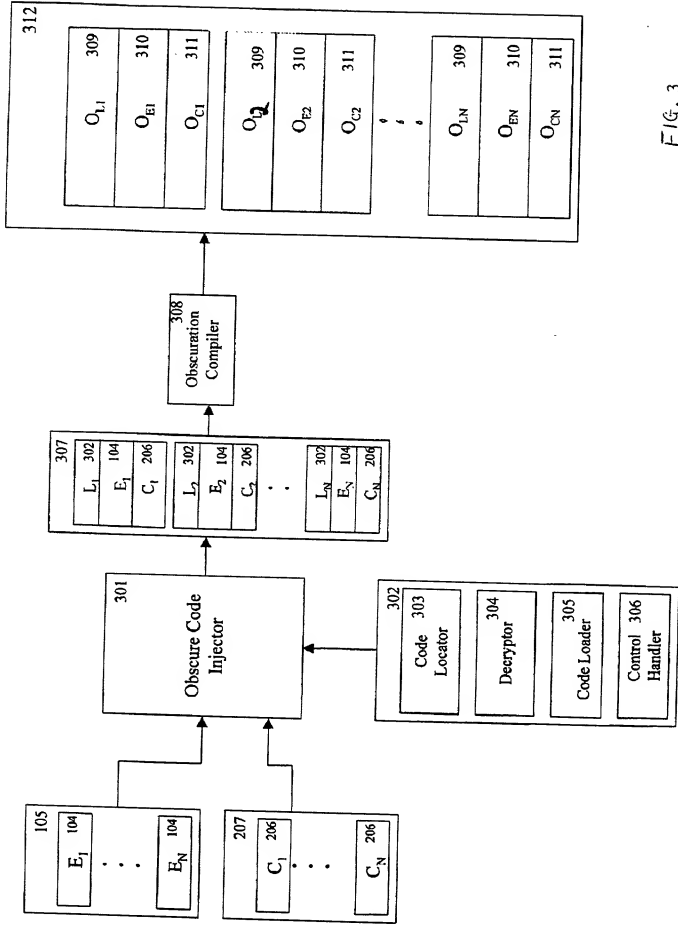


Fig. 2



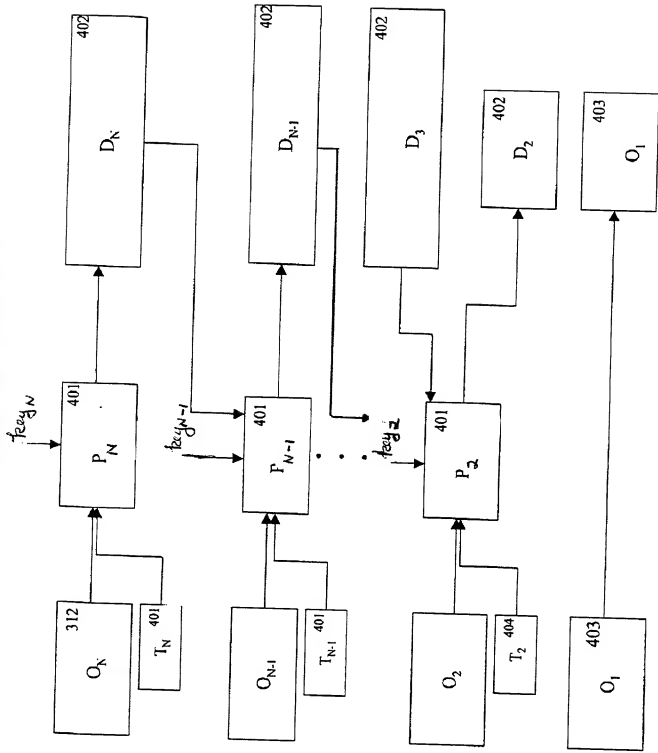
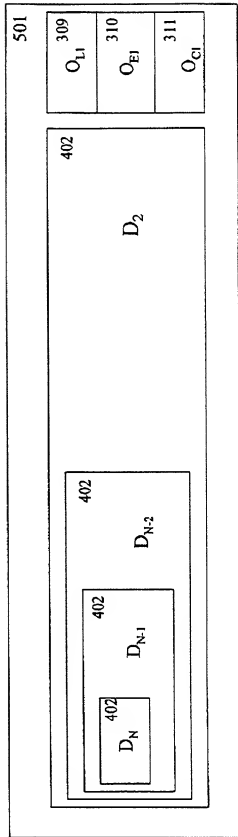


FIG. 4

000240*15625560



DAT FILE

FIG. 5

000240-15625560

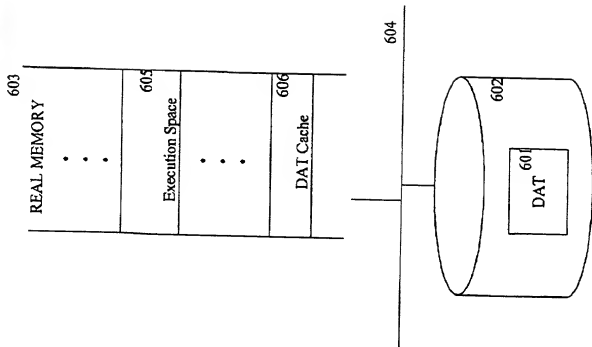


FIG. 6

000240*15625560

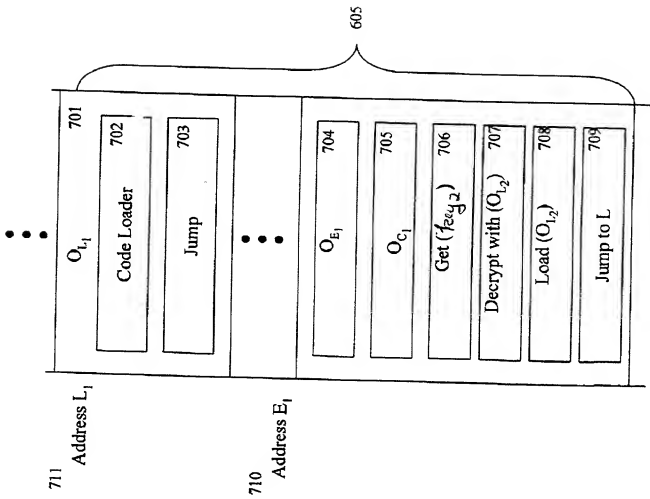


Fig. 7

711

Address L

Code Locator	802	O_{L_2} 801
Decryptor	803	
Code Loader	804	
Control Handler	805	
O_{E_2}	807	
O_{C_2}	808	
Get (f_{eq_3})	809	
Decrypt with (O_{L_3})	810	
Load (O_{L_3})	811	
Jump to L	812	

806

Address E_2

**DECLARATION
AND POWER OF ATTORNEY**

As a below named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below at 201 et seq. underneath my name.

I believe I am the original, first and sole inventor if only one name is listed at 201 below, or an original, first and joint inventor if plural names are listed at 201 et seq. below, of the subject matter which is claimed and for which a patent is sought on the invention entitled

METHOD AND APPARATUS TO PREVENT REVERSE ENGINEERING AND TAMPERING

and for which a patent application:

☒ is attached hereto and includes amendment(s) filed on *(if applicable)*

☐ was filed in the United States on as Application No. *(for declaration not accompanying application)*

with amendment(s) filed on *(if applicable)*

☐ was filed as PCT international Application No. on and was amended under PCT Article 19 on *(if applicable)*

I hereby state that I have reviewed and understand the contents of the above identified application, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information known to me to be material to patentability as defined in Title 37, Code of Federal Regulations, § 1.56.

I hereby claim foreign priority benefits under Title 35, United States Code, § 119(a)-(d) of any foreign application(s) for patent or inventor's certificate listed below and have also identified below any foreign application for patent or inventor's certificate having a filing date before that of the application on which priority is claimed:

EARLIEST FOREIGN APPLICATION(S), IF ANY, FILED PRIOR TO THE FILING DATE OF THE APPLICATION			
APPLICATION NUMBER	COUNTRY	DATE OF FILING (day, month, year)	PRIORITY CLAIMED
			YES <input type="checkbox"/> NO <input type="checkbox"/>
			YES <input type="checkbox"/> NO <input type="checkbox"/>

I hereby claim the benefit under Title 35, United States Code, § 119(e) of any United States provisional application(s) listed below.

APPLICATION NUMBER	FILING DATE

I hereby claim the benefit under Title 35, United States Code, § 120 of any United States application(s) listed below and, insofar as the subject matter of each of the claims of this application is not disclosed in the prior United States application in the manner provided by the first paragraph of Title 35, United States Code § 112, I acknowledge the duty to disclose information which is material to patentability as defined in Title 37, Code of Federal Regulations, § 1.56 which became available between the filing date of the prior application and the national or PCT international filing date of this application:

APPLICATION SERIAL NO.	FILING DATE	STATUS		
		PATENTED	PENDING	ABANDONED

POWER OF ATTORNEY: As a named inventor, I hereby appoint S. Leslie Misrock (Reg. No. 18872), Harry C. Jones, III (Reg. No. 20280), Berj A. Terzian (Reg. No. 20060), David Weid, III (Reg. No. 21094), Jonathan A. Marshall (Reg. No. 24614), Barry D. Rein (Reg. No. 22411), Stanton T. Lawrence, III (Reg. No. 25736), Charles E. McKenney (Reg. No. 22795), Philip T. Shannon (Reg. No. 24278), Francis E. Morris (Reg. No. 24615), Charles E. Miller (Reg. No. 24576), Gidon D. Stern (Reg. No. 27469), John J. Lauter, Jr. (Reg. No. 27814), Brian M. Poissant (Reg. No. 28462), Brian D. Coggio (Reg. No. 27624), Rory J. Radding (Reg. No. 28749), Stephen J. Harbulak (Reg. No. 29166), Donald J. Goodell (Reg. No. 19766), James N. Palik (Reg. No. 25510), Thomas E. Friebe (Reg. No. 29258), Laura A. Coruzzi (Reg. No. 30742), Jennifer Gordon (Reg. No. 30753), Allan A. Fanucci (Reg. No. 30256), Geraldine F. Baldwin (Reg. No. 31232), Victor N. Balancia (Reg. No. 31231), Samuel B. Abrams (Reg. No. 30605), Steven I. Wallach (Reg. No. 35402), Marcia H. Sundeen (Reg. No. 30893), Paul J. Zegger (Reg. No. 33821), Edmond R. Bannon (Reg. No. 32110), Bruce J. Barker (Reg. No. 32391), Adriane M. Antler (Reg. No. 32605), Thomas G. Rowan (Reg. No. 34419), James G. Markey (Reg. No. 31636), Thomas D. Kohler (Reg. No. 32797), Scott D. Stimpson (Reg. No. 33607), Gary S. Williams (Reg. No. 31066), William S. Galliani (Reg. No. 33885), Ann L. Gisolfi (Reg. No. 31956), Todd A. Wagner (Reg. No. 35399), Scott B. Familant (Reg. No. 35514), Kelly D. Talcott (Reg. No. 39582), Francis D. Cerreto (Reg. No. 38100), Anthony M. Insoigna (Reg. No. 35203), Brian M. Rothery (Reg. No. 35340), Brian D. Siff (Reg. No. 35679), and Alan Tenenbaum (Reg. No. 34939), all of Pennie & Edmonds LLP, whose addresses are 1155 Avenue of the Americas, New York, New York 10036, 1667 K Street N.W., Washington, DC 20006 and 3300 Hillview Avenue, Palo Alto, CA 94304, and each of them, my attorneys, to prosecute this application, and to transact all business in the Patent and Trademark Office connected therewith.

SEND CORRESPONDENCE TO:		PENNIE & EDMONDS LLP 1155 Avenue of the Americas New York, N.Y. 10036-2711		DIRECT TELEPHONE CALLS TO: PENNIE & EDMONDS LLP DOCKETING (212) 790-2803	
201	FULL NAME OF INVENTOR	LAST NAME ZIA	FIRST NAME Zhen	MIDDLE NAME	
	RESIDENCE & CITIZENSHIP	CITY Germantown	STATE OR FOREIGN COUNTRY Maryland	COUNTRY OF CITIZENSHIP People's Republic of China	
	POST OFFICE ADDRESS	STREET 11423 Stony Point Place	CITY Germantown	STATE OR COUNTRY Maryland	ZIP CODE 20876
202	FULL NAME OF INVENTOR	LAST NAME SHEN	FIRST NAME Ji	MIDDLE NAME	
	RESIDENCE & CITIZENSHIP	CITY Gaithersburg	STATE OR FOREIGN COUNTRY Maryland	COUNTRY OF CITIZENSHIP People's Republic of China	
	POST OFFICE ADDRESS	STREET 7 Talley Court	CITY Gaithersburg	STATE OR COUNTRY Maryland	ZIP CODE 20878
203	FULL NAME OF INVENTOR	LAST NAME	FIRST NAME	MIDDLE NAME	
	RESIDENCE & CITIZENSHIP	CITY	STATE OR FOREIGN COUNTRY	COUNTRY OF CITIZENSHIP	
	POST OFFICE ADDRESS	STREET	CITY	STATE OR COUNTRY	ZIP CODE
204	FULL NAME OF INVENTOR	LAST NAME	FIRST NAME	MIDDLE NAME	
	RESIDENCE & CITIZENSHIP	CITY	STATE OR FOREIGN COUNTRY	COUNTRY OF CITIZENSHIP	
	POST OFFICE ADDRESS	STREET	CITY	STATE OR COUNTRY	ZIP CODE
205	FULL NAME OF INVENTOR	LAST NAME	FIRST NAME	MIDDLE NAME	
	RESIDENCE & CITIZENSHIP	CITY	STATE OR FOREIGN COUNTRY	COUNTRY OF CITIZENSHIP	
	POST OFFICE ADDRESS	STREET	CITY	STATE OR COUNTRY	ZIP CODE
206	FULL NAME OF INVENTOR	LAST NAME	FIRST NAME	MIDDLE NAME	
	RESIDENCE & CITIZENSHIP	CITY	STATE OR FOREIGN COUNTRY	COUNTRY OF CITIZENSHIP	
	POST OFFICE ADDRESS	STREET	CITY	STATE OR COUNTRY	ZIP CODE

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issuing thereon.

SIGNATURE OF INVENTOR 201	SIGNATURE OF INVENTOR 202	SIGNATURE OF INVENTOR 203
Zhen ZIA	Ji SHEN	
DATE	DATE	DATE
SIGNATURE OF INVENTOR 204	SIGNATURE OF INVENTOR 205	SIGNATURE OF INVENTOR 206
DATE	DATE	DATE